13장. 고차함수와 인라인 함수

13.1. 고차함수

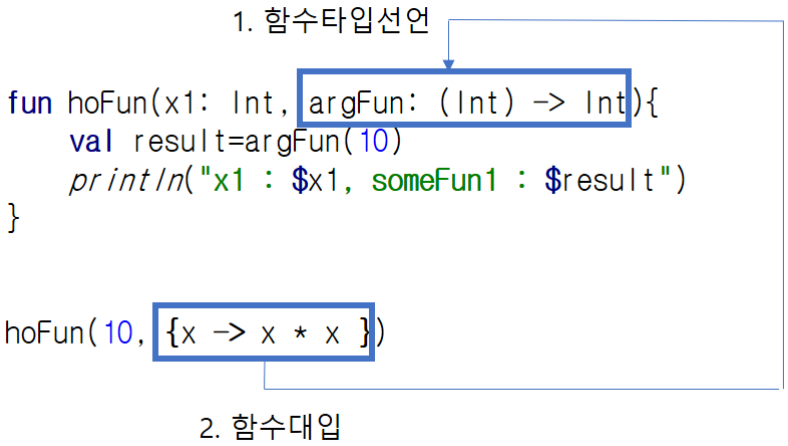
13.1.1. 고차함수란?

- 고차함수(High-Order Function, 고계함수)
- 함수의 매개변수로 함수를 전달 받거나 함수를 리턴시킬수 있는 함수를 지칭

```
fun normalFun(x1: Int, x2: Int): Int{
    return x1 + x2
}
```

```
fun hoFun(x1: Int, argFun: (Int) -> Int){
    val result=argFun(10)
    println("x1 : $x1, someFun1 : $result")
}

hoFun(10, {x -> x * x })
```



13.1. 고차함수

13.1.2. 고차함수와 함수 타입 매개변수

함수 타입의 매개변수 대입

- 함수 호출시 ( )을 생략가능

```
fun hoFun1(argFun: (Int) -> Int){
    val result=argFun(10)
    println("result : $result")
}
hoFun1({x -> x * x})
hoFun1 {x -> x* x }
```

```
val array= arrayOf(10, 20, 15, 22, 8)
array.filter{ x -> x > 10 }
        .forEach{ x -> println(x) }
```

```
inline fun IntArray.filter(
    predicate: (Int) -> Boolean
): List<Int>
```

13.1. 고차함수

```
fun hoFun_1(no: Int, argFun1: (Int)->Int, argFun2: (Int)->Boolean){

}
hoFun_1(10, {it * it}, {it > 10})
hoFun_1(10, {it * it}) {it > 10}
hoFun_1(10){it * it} {it > 10}//error
```

함수타입 디폴트값 이용

```
fun some(x1:Int = 10){
    println(x1)
}
some()
```

```
fun hoFun2(
        x1: Int,
        argFun1: (Int) -> Int,
        argFun2: (Int) -> Boolean = { x: : Int -> x > 10}
){
    val result = argFun1(x1)
    println("result : ${argFun2(result)}")
}

hoFun2(2, { x: Int -> x * x }, {x: Int -> x > 20})
hoFun2(4, { x: Int -> x * x })
```

13.1. 고차함수

13.1.3. 고차함수와 함수반환

```
fun hoFun5(str: String): (x1: Int, x2: Int) -> Int {
    when (str){
        "-" -> return { x1, x2 -> x1 - x2 }
        "*" -> return { x1, x2 -> x1 * x2 }
        "/" -> return { x1, x2 -> x1 / x2 }
        else -> return { x1, x2 -> x1 + x2 }
    }
}

val resultFun=hoFun5("*")
println("result * : ${resultFun(10, 5)}")
```

13.1. 고차함수

13.1.4. 함수 참조와 익명 함수 이용

함수 참조를 이용한 함수 전달

```
fun hoFun6(argFun: (x: Int) -> Int){
    println("${argFun(10)}")
}

hoFun6 { it * 5 }
```

```
fun hoFun6(argFun: (x: Int) -> Int){
    println("${argFun(10)}")
}

fun nameFun(x: Int): Int {
    return x * 5
}
hoFun6(::nameFun)
```

13.1. 고차함수

익명함수를 이용한 함수 전달

- 람다함수는 이름이 없는 함수
- 람다함수는 return 예약어를 이용하여 리턴 값을 명시할수 없다.

```
val lambdasFun={ x: Int ->
    println("i am lambdas function")
    return x * 10//error
}
```

- Anonymous Function은 단어뜻 그대로 이름이 없는 함수.

```
val anonyFun1 = fun(x: Int): Int = x * 10

val anonyFun2 = fun(x: Int): Int {
    println("i am anonymous function")
    return x * 10
}
```

```
fun hoFun7(argFun: (Int)->Int){
   println("${argFun(10)}")
}
hoFun7(fun(x: Int): Int = x * 10)
```

13.1. 고차함수

- **일반함수**

```
fun myFun(x: Int): Int {
    return x * 10
}
```

```
fun 함수명(매개변수): 리턴타입 {
    함수내용
}
```

- **Anonymous Function**

```
fun(x: Int): Int {
    return x * 10
}
```

```
fun 함수명(매개변수): 리턴타입 {
    함수내용
}
```

⇨

```
fun(매개변수): 리턴타입 {
    함수내용
}
```

- **Lambdas Function**

```
{ x: Int -> x * 10}
```

```
fun 함수명(매개변수): 리턴타입 {
    함수내용
}
```

⇨ `{ 매개변수 -> 함수내용 }`

13.1. 고차함수

13.1.5. 코틀린 API의 유용한 고차함수

함수 참조를 이용한 함수 전달

run()

- 단순 람다함수를 실행시키고 그 결과 값을 획득
- 객체의 맴버에 접근

```
inline fun <R> run(block: () -> R): R
```

```
val result= run {
    println("lambdas function call...")
    10 + 20
}
println("result : $result")
```

13.1. 고차함수

```
inline fun <T, R> T.run(block: T.() -> R): R
```

```
class User() {
    var name: String?=null
    var age: Int?=null

    fun sayHello(){
        println("hello $name")
    }
    fun sayInfo(){
        println("i am $name, $age years old")
    }
}

val user = User()
user.name="kkang"
user.age=33
user.sayHello()
user.sayInfo()
```

```
val runResult=user.run {
    name="kim"
    age=28
    sayHello()
    sayInfo()
    10 + 20
}

println("run result : $runResult")
```

13.1. 고차함수

apply()

- apply() 함수는 run() 함수와 사용 목적은 동일한데 리턴되는 값에 차이

- run() 함수의 리턴 값은 대입된 람다함수의 리턴값이 그대로 run() 함수의 리턴값
- apply() 함수는 applay() 함수를 적용한 객체가 리턴

```
inline fun <T> T.apply(block: T.() -> Unit): T
```

```
val user3=user.apply {
    name="park"
    sayHello()
    sayInfo()
}
println("user name : ${user.name}, user3 name : ${user3.name}")
user.name="aaa"
user3.name="bbb"
println("user name : ${user.name}, user3 name : ${user3.name}")
```

13.1. 고차함수

let()

- let 을 이용하는 객체를 let 의 매개변수로 지정한 람다함수에 매개변수로 전달

```
inline fun <T, R> T.let(block: (T) -> R): R
```

```
class User() {

    var name: String?=null
    var age: Int?=null

    constructor(name: String, age: Int) : this() {
        this.name=name
        this.age=age
    }
}fun letTestFun(user: User){
    println("letTestFun() : ${user.name} .. ${user.age}")
}
val user4=User("kkang", 33)
letTestFun(user4)
```

```
User("kim", 28).let { user ->
    letTestFun(user)
}
```

13.1. 고차함수

with()

- with() 함수는 run() 함수와 사용 목적이 유사
- 객체의 맴버들을 반복적으로 접근할 때 객체명을 일일이 명시하지 않고 멤버들을 바로 이용하기 위한 용도
- run() 함수는 run() 함수를 이용한 객체가 람다함수에서 바로 이용
- with() 함수는 with() 함수의 매개변수로 지정한 객체를 람다함수에서 이용

```
inline fun <T, R> with(receiver: T, block: T.() -> R): R
```

```
user.run {
    name="kkang"
    sayHello()
}

with(user){
    name="kkang"
    sayHello()
}
```

13.2. 인라인 함수

13.2.1. 인라인 함수란?

- 고차함수의 런타임시 성능이슈 문제

```
fun hoFunTest(argFun: (x1: Int, x2: Int) -> Int){
    argFun(10, 20)
}
fun main(args: Array<String>) {
    val result = hoFunTest { x1, x2 -> x1 + x2 }
}
```

```
public static final void hoFunTest(@NotNull Function2 argFun) {
   Intrinsics.checkParameterIsNotNull(argFun, "argFun");
   argFun.invoke(Integer.valueOf(10), Integer.valueOf(20));
}

public static final void main(@NotNull String[] args) {
   Intrinsics.checkParameterIsNotNull(args, "args");
   hoFunTest((Function2)null.INSTANCE);
}
```

13.2. 인라인 함수

- inline 함수는 함수 선언 앞에 inline 예약어를 추가한 함수
- 컴파일 단계에서 정적으로 포함

```
inline fun hoFunTest(argFun: (x1: Int, x2: Int) -> Int){
    argFun(10, 20)
}
fun main(args: Array<String>) {
    hoFunTest { x1, x2 -> x1 + x2 }
}
```

```
public static final void main(@NotNull String[] args) {
   //.............
   int x2 = 20;
   int x1 = 10;
   int var10000 = x1 + x2;
}
```

13.2. 인라인 함수

13.2.2. 노인라인

- noinline 예약어를 이용하여 inline에 적용되는 람다함수와 적용되지 않는 람다함수를 구분

```
inline fun inlineTest(argFun1: (x: Int) -> Int, argFun2: (x: Int) -> Int){
    argFun1(10)
    argFun2(10)
}
fun main(args: Array<String>) {
    inlineTest({it * 10}, {it * 20})
}
```

```
public static final void main(@NotNull String[] args) {
   Intrinsics.checkParameterIsNotNull(args, "args");
   int it = 10;
   int var10000 = it * 10;
   it = 10;
   var10000 = it * 20;
}
```

13.2. 인라인 함수

- noinline 예약어를 이용하여 inline에 포함하지 않아도 되는 함수타입의 매개변수를 명시적으로 선언

```
inline fun inlineTest(argFun1: (x: Int) -> Int, noinline argFun2: (x: Int) -> Int){
    argFun1(10)
    argFun2(10)
}
fun main(args: Array<String>) {
    inlineTest({it * 10}, {it * 20})
}
```

```
public static final void main(@NotNull String[] args) {
   Intrinsics.checkParameterIsNotNull(args, "args");
   Function1 argFun2$iv = (Function1)null.INSTANCE;
   int it = 10;
   int var10000 = it * 10;
   argFun2$iv.invoke(Integer.valueOf(10));
}
```

13.2. 인라인 함수

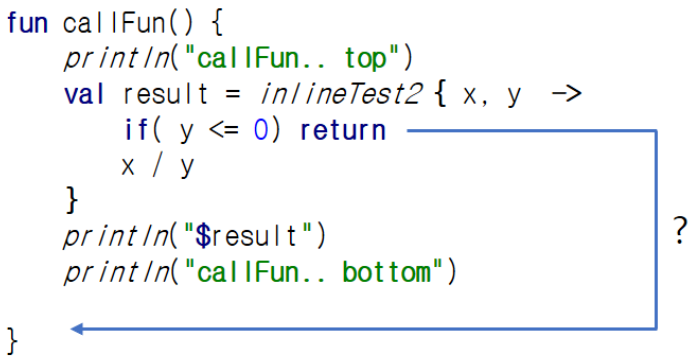
13.2.3. 논로컬 반환

- Non-local return 이란 람다함수 내에서 람다함수를 포함하는 함수를 벗어나게 하고자 하는 기법
- 코틀린에서는 return 을 이름이 정의된 일반함수와 Anonymous Function 에서만 사용이 가능하며 람다함수에서는 사용 불가

```
fun inlineTest2(argFun: (x: Int, y: Int) -> Int): Int{
    return argFun(10, 0)
}

fun callFun() {
    println("callFun.. top")
    val result = inlineTest2 { x, y ->
        if( y <= 0) return//error
        x / y
    }
    println("$result")
    println("callFun.. bottom")
}
```

13.2. 인라인 함수



- 고차함수가 inline 으로 선언되어 있다면 이 고차함수에 적용되는 람다함수 내에는 return 구문을 사용 가능

```
inline fun inlineTest2(argFun: (x: Int, y: Int) -> Int): Int{
    return argFun(10, 0)
}
fun callFun() {
    println("callFun.. top")
    val result = inlineTest2 { x, y ->
        if( y <= 0) return
        x / y
    }
    println("$result")
    println("callFun.. bottom")
}
fun main(args: Array<String>) {
    callFun()
}
```

13.2. 인라인 함수

크로스 인라인

- inline 되는 고차함수라고 하더라도 대입되는 람다함수가 고차함수 내에서 다른 객체에 대입되어 사용이 된다면 return 사용에 제약

```
open class TestClass {
    open fun some() {}
}
fun inlineTest3(argFun: () -> Unit){
    val obj = object : TestClass() {
        override fun some() = argFun()
    }
}
```

```
open class TestClass {
    open fun some() {}
}
inline fun inlineTest3(argFun: () -> Unit){
    val obj = object : TestClass() {
        override fun some() = argFun()//error
    }
}
```

13.2. 인라인 함수

- crossinline 은 함수가 inline 으로 선언되었다고 하더라도 대입되는 람다함수에 return 이 사용되지 않게 하기 위한 예약어

```
inline fun inlineTest3(crossinline argFun: () -> Unit){
    val obj = object : TestClass() {
        override fun some() = argFun()
    }
}

fun crossInlineTest(){
    println("aaa")
    inlineTest3 {
        return//error
    }
}
```

13.2. 인라인 함수

13.2.4. 라벨로 반환

- 람다에서 return 을 이용하여 자신이 대입되는 고차함수의 수행을 끝내야 하는 경우

```
val array = arrayOf(1,-1, 2)
fun arrayLoop() {
    println("loop top..")
    array.forEach {
        println(it)
    }
    println("loop bottom..")
}
arrayLoop()
```

실행결과

```
loop top..
1
-1
2
loop bottom..
```

13.2. 인라인 함수

- 람다함수에 return 구문을 추가

```
val array = arrayOf(1,-1, 2)
fun arrayLoop() {
    println("loop top..")
    array.forEach {
        if(it < 0) return
        println(it)
    }
    println("loop bottom..")
}
arrayLoop()
```

실행결과

```
loop top..
1
```

13.2. 인라인 함수

- 데이터를 출력하는 람다함수만 벗어나게 하고 싶은 경우

```
val array = arrayOf(1,-1, 2)
fun arrayLoop() {
    println("loop top..")
    array.forEach aaa@{
        if(it < 0) return@aaa
        println(it)
    }
    println("loop bottom..")
}
arrayLoop()
```

실행결과

```
loop top..
1
2
loop bottom..
```

13.2. 인라인 함수

- 자동으로 고차함수에 label이 추가되고 그 이름을 이용 가능

```
val array = arrayOf(1,-1, 2)
fun arrayLoop() {
    println("loop top..")
    array.forEach {
        if(it < 0) return@forEach
        println(it)
    }
    println("loop bottom..")
}
arrayLoop()
```

실행결과

```
loop top..
1
2
loop bottom..
```

13.2. 인라인 함수

```
inline fun hoTest(argFun: (String)->Unit){
    argFun("hello")
    argFun("kim")
    argFun("kkang")
}
fun labelTest(){
    println("test top....")
    hoTest {
        if(it.length < 4) return@hoTest
        println(it)
    }
    println("test bottom....")
}
```

실행결과

```
test top....
hello
kkang
test bottom....
```

13.3. 클로저

- 클로저는 프로그래밍 언어에서 흔히 함수의 스코프(scope)에 사용된 변수를 바인딩하기 위한 기법
- 함수가 호출될 때 발생하는 데이터를 함수 호출후에도 그대로 유지되어 이용할수 있게 하기 위한 기법

```
fun closureTest(x: Int){
    println("argument $x")
}
```

```
fun closureTest(x: Int): (Int)->Int{
    println("argument $x")
    return { it * 10 }
}

fun main(args: Array<String>) {
    val someFun1= closureTest(2)
    val someFun2= closureTest(3)

    println("${someFun1(10)}")
    println("${someFun2(10)}")
}
```

실행결과

```
argument 2
argument 3
100
100
```

13.3. 클로저

```
public static final Function1 closureTest(int x) {
    String var1 = "argument " + x;
    System.out.println(var1);
    return (Function1)null.INSTANCE;
}
```

13.3. 클로저

```
fun closureTest(x: Int): (Int)->Int{
    println("argument $x")
    return { it * x }
}

fun main(args: Array<String>) {
    val someFun1= closureTest(2)
    val someFun2= closureTest(3)

    println("${someFun1(10)}")
    println("${someFun2(10)}")
}
```

```
public static final Function1 closureTest(final int x) {
   String var1 = "argument " + x;
   System.out.println(var1);
   return (Function1)(new Function1() {
      // $FF: synthetic method
      // $FF: bridge method
      public Object invoke(Object var1) {
         return this.invoke(((Number)var1).intValue());
      }

      public final int invoke(int it) {
         return it * x;
      }
   });
}
```